Coqcots & Pycots: non-stopping components for safe dynamic reconfiguration

Jérémy Buisson, Everton Calvacante, Fabien Dagnat, Elena Leroux, Sébastien Martinez

Coqcots & Pycots: non-stopping components for safe dynamic reconfiguration

Jérémy Buisson, Everton Cavalcante, Fabien Dagnat,
Elena Leroux, Sébastien Martinez

April 28, 2014

Abstract

Software systems have to face evolutions of their running context and users. Therefore, the so-called *dynamic reconfiguration* has been commonly adopted for modifying some components and/or the architecture at runtime. Traditional approaches typically stop the needed components, apply the changes, and restart the components. However, this scheme is not suitable for critical systems and degrades user experience. This paper proposes to switch from the stop/restart scheme to *dynamic software updating* (DSU) techniques. Instead of stopping a component, its implementation is replaced by another one specifically built to apply the modifications while maintaining the best quality of service possible. The major contributions of this work are: (i) the integration of DSU techniques in a component model, and; (ii) a reconfiguration development process including specification, proof of correctness using Coq, and a systematic method to produce the executable script. In this perspective, the use of DSU techniques brings higher quality of service when reconfiguring component-based software and the formalization allows ensuring the safety and consistency of the reconfiguration process.

1 Introduction

Software systems need to be highly available and should be built using secure, safe, performant, and robust components. These components must be regularly modified to fix vulnerabilities and bugs, to face new environments, and to offer new services. Enabling these evolutions, while maintaining a high level of availability, requires changing the architecture of such a system during its execution. This capability is especially important for critical systems such as air-traffic control systems and networks, in which stopping systems is not an option due to financial or human costs. It also improves user experience as a user can continue to use a software system being updated without noticing the update, *i.e.*, the updating process is transparent to the users.

In software engineering, *dynamic reconfiguration* was introduced to build component-based software systems that can be modified during their execution, with minimal or no interruption. In this approach, components and connectors of a system can be inserted, removed or replaced at runtime, thus fostering the continuity of the services it provides.

Since the proposition of the *quiescence* concept [9], reconfiguring a system typically requires the suspension of a set of components that will be affected by the reconfiguration. Maintaining the system in an operational status while stopping part of its components leads to a visible degradation of its quality of service [6, 10] due to component dependencies. Another essential issue to be considered is to preserve the consistency of the component assembly throughout the reconfiguration process. Some works in the literature have faced these challenges by minimizing the set of suspended components and/or decreasing the duration of their suspension [7, 18]. Focusing on consistency, Boyer et al. [2] propose a scheme in which an invariant requires to stop any component that depends on a stopped component. However, such dependencies often propagate until the user frontend, and then the system may be almost entirely stopped.

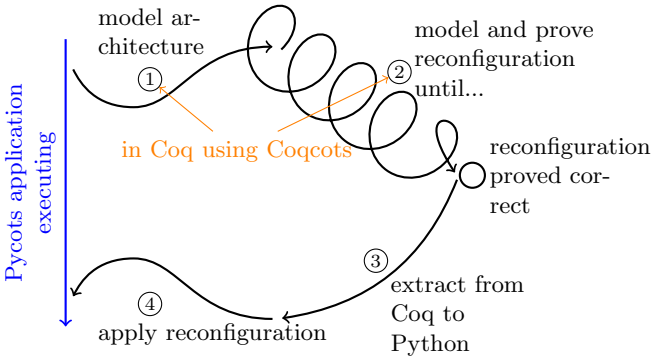

Figure 1: Overview of the proposed approach.

To maintain service continuity, we must refrain from stopping components when reconfiguring a system. In this paper, we propose to use *dynamic software updating* (DSU) techniques [14, 16] instead of suspending components. The main idea is to mitigate the effect of any reconfiguration action by dynamically updating the implementation of directly and indirectly affected components. For example, if a component B used by a component A needs to be reconfigured, then the component A can be updated with a new implementation that no longer uses B before reconfiguring B.

The two major contributions of this paper are: (i) integration of DSU techniques in a component model named Coqcots/Pycots, and (ii) a reconfiguration development process. In Figure 1, which depicts the process, the left-hand side arrow represents the execution flow of the target application. During execution, if a reconfiguration is needed, its design and execution follow the four steps on the right of the figure. The two first steps of this process are performed in Coq using Coqots: ① the current architecture of the target software system is modeled as a Coqcots architecture, and ② this model is used as an input to develop the reconfiguration using the Coq proof assistant. Once the reconfiguration is valid: ③ the reconfiguration script is translated to Python and ④ the Python script is submitted to the Pycots manager, which is a platform component that receives and applies reconfiguration scripts.

The remainder of this paper covers steps ①, ② and ④ of the process. Section 2 outlines the DSU techniques. Section 3 presents Coqcots, our component model for dynamic reconfiguration, while Section 4 covers Pycots, the Python framework that implements it. Section 5 is dedicated to the validation of our approach with a case study. Section 6 briefly discusses some relevant work related to the dynamic reconfiguration of component-based systems. Finally, Section 7 contains final remarks and directions for future work. All the material described in this paper is available online at `http://coqcots.gforge.inria.fr`.

2 Dynamic software updating

Dynamic software updating (DSU) gathers several mechanisms whose goal is to update software at runtime. Whereas dynamic reconfiguration affects the architectural structure, DSU modifies the software at procedural or object-oriented level. Most classical DSU mechanisms replace a function by its newer version when it is not being called (*i.e.*, when it is not in the call stack) and convert instances of a given class to a new structure by using a transformer function.

Surveys [14, 16] list several platforms that provide DSU mechanisms. Each of these platforms offers a different perspective on DSU among the following aspects: How and when shall the update be triggered? How and when shall data be converted? How shall the execution flow of the application be modified? In this work we use Pymoult [13], a Python-based platform that provides several of the existing DSU mechanisms and well addresses all of the previously mentioned aspects.

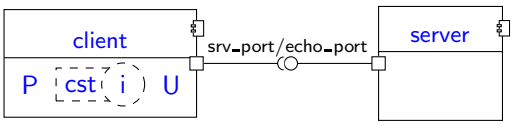

Figure 2: Anatomy of a Coqcots architecture.

In the current state-of-the-art, two triggering mechanisms are proposed. With *active* triggering, the program signals to the platform that it can be updated by calling a specific function placed in the code by the program developer. With *passive* triggering, the platform detects when the program can be updated. In both cases, detecting this moment requires being able to monitor the program (*e.g.*, detect if a function is in the stack), especially when using passive triggering. Pymoult provides the two triggering mechanisms as well as tools for monitoring the program.

Data is usually converted by the combination of an access strategy (*eager* or *lazy*) and a conversion method. For the eager strategy, Pymoult intercepts the creation of each object in order to store a weak reference to this object. The created pool is then used to access all objects at a given time. For the lazy strategy, Pymoult uses the Python meta-object protocol to access objects whenever their fields are used (read or written). Regarding conversion, Pymoult provides tools such as the usage of *mixins* or *proxies*.

Several ways of modifying the execution flow (*e.g.*, changing the code of a function, altering the stack) are used in the literature. Pymoult supports *safe redefinition* for modifying a function or method code when it is not in the stack as well as more sophisticated techniques such as *stack reconstruction* [11] and *thread rebooting* [5, 13]. The two latter mechanisms allow to change the behavior of a function even if it is in the stack. In order to initialize the new behavior, it is possible to capture the runtime stack and read values from it thanks to the Python introspection tools.

3 The Coqcots component model

In this section we introduce our component model named Coqcots, formally defined by using the Coq formalism. The design of Coqcots follows the consensual component model described by Boyer et al. [2], except for the following points. In [2], each component follows a definite lifecycle: components can be started or stopped, and for each state of the lifecycle, a specific behavior is defined and embedded in the component. Instead of the predefined collection of behavior variants, Coqcots uses DSU techniques to dynamically change the behavior as needed by each reconfiguration. Moreover, changing the behavior of a component implies a change of its type. For example, a port may be added or removed when a new behavior implements a different set of functionalities. This change of type is explicit in Coqcots.

In the following, we first introduce our component model and explain how to specify a system architecture. Next, we define five operations used to reconfigure a system. Finally, we introduce a set of invariants whose role is to ensure a correct definition of a system architecture.

3.1 Specification of an architecture

As illustrated in Figure 2, a component (client) has an associated implementation (i) that uses a set of *used services* (U, which contains only the srv_port service in the case of the client component) to provide a set of *provided services* (P, which is empty for the client component). The architecture instance contains components (client and server) and defines the bindings between a used service of one of its components and a provided service of another of its components (srv_port/echo_port). When a used service of a component is not bound, the implementation of that component cannot use this unbound service.

In our approach, the implementation of a component may come with a precondition that specifies its assumption on the architecture in which the component is instantiated. This *architectural constraint* (cst for the client component) is an invariant that ensures that the implementation will only run in the expected situation. For example, a given component may require that some of its used services must be bound (the *mandatory services*), while the other services are allowed to remain unbound (the *optional services*). The component implementation can then assume that only optional services may be unbound. Using architectural constraints enables the designer to assume the mandatory semantics discussed by Boyer et al. [2] and Bruneton et al. [4].

Formally, Coqcots comes with two predicates, *contains* and *binds*, which respectively state that an architecture contains a given component and a given binding. The *contains* predicate takes as first argument an architecture a, followed by the five elements of a component: (1) the name of the component c, (2) the set[1] of its used services U, (3) the set of its provided services P, (4) its architectural constraint cst, and (5) its implementation i. The definition of *contains* is:

```
Parameter contains:
    ∀ (a: arch) (c: comp) (U: facet) (P: facet)
      (cst: arch → comp → Prop)
      (i: ∀ (u: facet_record U),
            cst a c → no_exc_if_bound a c u → facet_record P),
      Prop.
```

The architectural constraint is defined as a function that maps an architecture a and a component c to a proposition. It is used by the implementation as a precondition to provide the component's services (term cst a c). The other precondition no_exc_if_bound relieves the implementation from defensively checking for availability when a used service is guaranteed to be bound according to the constraint cst.

The *binds* predicate formally states that a binding exists in an architecture a. The binding is represented by six values: three for the client component (the user of the binding) and three for the server component (the provider of the binding). Both components are represented by (1) their identity respectively clt and srv, (2) the set of their involved services clt_U and srv_P, and (3) the ports clt_port and srv_port bound by the predicate. The definition of *binds* is:

```
Parameter binds:
    ∀ (a: arch)
      (clt: comp) (clt_U: Type) (clt_port: namedport clt_U)
      (srv: comp) (srv_P: Type) (srv_port: namedport srv_P),
      Prop.
```

Using these two predicates, the designer can define an architecture. For example, the architecture represented in Figure 2 is an element of the following type:

```
Definition client_server :=
  { a | ∃ client server,
      contains a client client_use_facet client_provide_facet
          client_constraint (client_implementation a client)
      ∧ contains a server server_use_facet server_provide_facet
          server_constraint (server_implementation a server)
      ∧ binds a client _ srv_port server _ echo_port }.
Definition client_constraint (self_arch: arch) (self: comp) :=
      ∃ s provs port, binds self_arch self _ srv_port s provs port.
Definition server_constraint (self_arch: arch) (self: comp) :=True.
```

In this exemple, the client srv_port must be bound as it is a mandatory dependency, and the server has no constraint. For the binding, the set of used and provided ports (third and sixth arguments elided as _) are inferred by Coq.

[1] In our model, we define a set of ports as a **facet**.

3.2 Reconfiguration operations

In this subsection, we formally define five primitive reconfiguration operations:

1. *create* adds a new component to the current architecture by taking its used and provided ports, constraint and implementation.
2. *destroy* removes an existing component from the current architecture by taking the name of the component.
3. *link* creates a binding from a provided ports of a component to a used ports of another component by taking the requiring component and its used port and the providing component and its provided port.
4. *unlink* destroys a binding from the current architecture by using the same parameters as *link*.
5. *hotswap* changes the behavior of an existing component by taking the component's name, the four new elements of the component, and two functions mapping respectively the used ports and the provided ports of previous version to the ones of the new version.

For sake of space, we only explain in details the *create* operation, whose Coq code is quite compact and easy to understand. All the other operations follow a similar scheme.

The *create* operation is a function that returns a pair r composed of the new architecture new_a and the newly created component new_c. They satisfy the create_post postcondition informally described later in this section. The *create* function takes seven parameters: (1) the current architecture a, (2) the set of used services of the new component U, (3) a proof U_all_opt that the used services are all of the type **optional**[2], (4) the set of provided services P, (5) the constraint cst, (6) the implementation i, and (7) a proof cst_all_hold that in the resulting architecture the architectural constraints of all the components (including the created one) are satisfied. The *create* operation is defined by:

```
Parameter create:
  ∀ (a: arch) (U: facet)
    (U_all_opt: List.Forall (fun p ⇒ ∃ t, p = optional t)
                            (ports_of _ (facet_spec U)))
    (P: facet) (cst: arch → comp → Prop)
    (i: ∀ self_arch self u, cst self_arch self
        → no_exc_if_bound self_arch self u → facet_record P)
    (cst_all_hold: ∀ new_a new_c,
        create_post a U U_all_opt P cst i new_a new_c
        → ∀ c' U' P' (cst': arch → comp → Prop)
            (i': ∀ a'' c'' u, cst' a'' c''
                 → no_exc_if_bound a'' c'' u → _),
          contains new_a c' U' P' cst' (i' new_a c')
          → cst' new_a c'),
    { r: arch × comp | let (new_a, new_c) := r in
      create_post a U U_all_opt P cst i new_a new_c }.
```

The postcondition of the *create* operation is the conjunction of six parts: (1) The previous architecture a does not contain the newly created component new_c. (2) The new component new_c exists in the new architecture new_a with the given elements (used U and provided P services, constraint cst and implementation i). (3) The new architecture new_a contains all the components of the previous architecture a. (4) The new architecture new_a contains only the components contained by the previous architecture a and the new component new_c. (5) In the new architecture new_a, the new component new_c is well-defined, *i.e.* it has unique elements (U, P, cst and i). (6) The previous a and new new_a architectures contain exactly the same bindings.

[2]The **optional** type models that used services can be unbound, thus having no value.

It is important to notice that a reconfiguration operation should preserve constraints and bindings of unaffected components. We address this point with the so-called *frame axiom* approach [8] in postcondition of reconfiguration operations. For example, regarding the *create* operation, postconditions 3, 4 and 6 above are the frame axioms.

The *hotswap* operation is the key point of Coqcots. The goal of this operation is to replace ports, constraints and implementation of a given component. It is important to notice that all of these replacements must be done at once. This constraint of our model comes from the fact that the type of an implementation depends on ports and constraints. For example, changing a port while keeping the implementation unchanged, is not well-typed. For the same reason, the bindings must also be adjusted. To do so, the *hotswap* operation takes two additional parameters, which map bound ports of the old set of ports to the new set of ports. As these mappings are restricted to bound ports, it is possible to remove ports as long as they are not bound in the architecture. Our definition of the *hotswap* operation is therefore consistent with *contextual substitutability* as defined by Brada [3].

The *hotswap* operation replaces the classical start/stop operations. With this operation, the developer is able to ensure the best continuity of service during a reconfiguration. Indeed, (s)he is able to define a new behavior for the component by providing partial services or all of its services by using other providers for its used services. Notice that this is possible even if the component initial design has not anticipated the situation. Moreover, behavioral changes must consistently reflect in the type of the component, so that any service degradation is explicit in the component type. This makes service degradation controllable.

3.3 Invariants

Coqcots is equipped with a set of invariants to ensure the soundness of the architecture definition:

Correct typing of bindings. This invariant excludes all architectures containing at least two ports bound together having incompatible types.

Existence of bound components. This invariant checks that bound components belong to the architecture.

Unicity of used service bindings. This invariant checks that the architecture does not contain a used port bound to two different provided ports.

Unicity of components characteristics. This invariant ensures that, for each component, the set of ports, constraints, and implementation are defined only once.

Satisfaction of component constraints. This invariant ensures that for each component, its constraint holds.

An architecture is *consistent* if the five previously defined invariants hold. We have proved that any architecture, obtained by applying Coqcots reconfiguration operations starting from the empty architecture, is consistent. It relies on two sub-proofs: (1) the empty architecture is consistent and (2) any of the five proposed reconfiguration operations preserves the invariants and then consistency. These proofs are specified using about 2500 lines of Coq code.

4 The Pycots Framework

In order to study practical issues, we have implemented Pycots, a Python-based framework that implements Coqcots. This section contains a brief description of Pycots.

The Pycots framework is composed of (1) a `Component` class, which is used to encapsulate components into black boxes, and (2) functions for reconfiguration operations. A component is basically an object that encapsulates its implementation, which is also an object. Each port is a method of this implementation object, which is either injected by the framework (used port) or coded by the developer (provided port). The provided ports are also exposed as public methods of the component object.

For each used port, we generate a *proxy*, whose role is to redirect method calls to their destinations. The advantage of the DSU approach is that it is easy to perform the *link* and *unlink* operations as they simply consist in

hotswapping the proxy implementation of the used port between the two alternative implementations: (1) when the used port is bound, the proxy forwards method calls to the provider; (2) otherwise, the proxy raises an exception.

It is important to notice that the Pycots framework is relieved from runtime verification of constraints, typing and invariants. Indeed, we assume that these issues have been proved with Coq. Furthermore, we split the *hotswap* operation into several more primitive operations that add or remove ports to a component. These operations simply inject or remove proxy methods in the component. In order to hotswap a component implementation, we rely on Pymoult [13], which offers several DSU mechanisms as described in Section 2. The manager, which listens for and executes reconfigurations, is borrowed from Pymoult too.

5 Validation

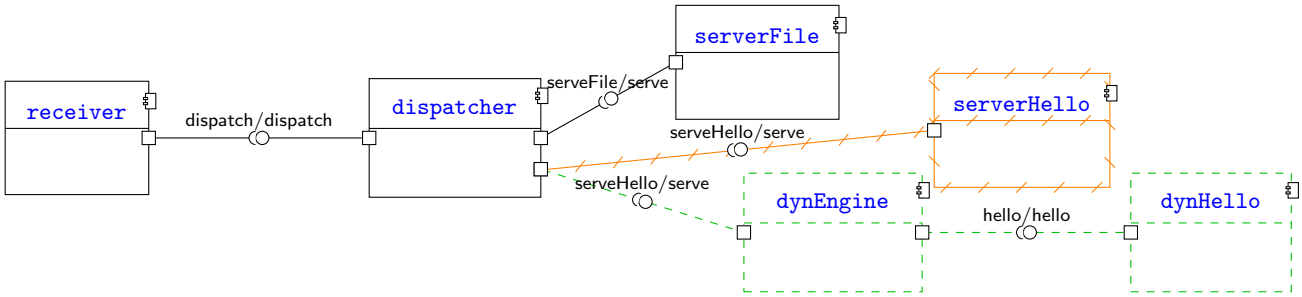

Figure 3: The software architecture of the web server after dynamic reconfiguration.

To validate our approach, we have studied a simple web server whose architecture is depicted in Figure 3. Strips denote the bindings and components that are removed by the reconfiguration, whereas dashes indicate the bindings and components that are added during reconfiguration.

The architecture initially contains four components: (1) the `receiver` wraps an instance of `BaseHTTPServer`, which receives and decodes HTTP requests; (2) the `dispatcher` dispatches requests to handlers according to the requested URL; (3) `serverHello` generates a dynamic web page with a "Hello, world" greeting, and; (4) `serverFile` detects that the given URL is a file name whose content is sent as a response. Its Coqcots model is the conjunction of 44 facts.

The proposed reconfiguration splits the `serverHello` component into two components: `dynEngine`, a generic engine that generates dynamic pages, and `dynHello`, the greetings handler. Since the `serverFile` component is not affected by this reconfiguration, it will continue to handle requests during the reconfiguration. The main idea is to temporarily hotswap the implementation of the `dispatcher` component such that it continues to serve the requests targeting the `serverFile` component while it enqueues those for `serverHello`. The main steps of this reconfiguration are:

1. To modify the implementation of the `receiver` component. As `BaseHTTPServer` handles all the requests by using a single thread, we use `ThreadingMixIn`, a mixin class from the standard library that transforms `BaseHTTPServer` such that it spawns a new thread for each request. Therefore, it will be possible to suspend a thread to delay a request.

2. The implementation of the `dispatcher` component is replaced by the following one: (1) it suspends the current thread (the request-handler thread) by using a global event object, if it receives a request for the `serverHello` component, and (2) it works as before, if a request for the `serverFile` component is received.

3. Once the two previous steps are completed, the serveHello port of `dispatcher` is no longer used: the web server is ready for the architectural changes. The binding between the `dispatcher` and `serverHello` components

is removed, the `dynEngine` and `dynHello` components are instantiated and bound, and then `dynEngine` is hotswapped to add its provided port and `dispatcher` is bound to `dynEngine`.

4. Finally, `dispatcher` is hotswapped back to its initial implementation and suspended threads are resumed.

The Coqcots model of this reconfiguration contains about 100 lines of Coq. It proves that the reconfiguration is correct and that requests targeting the `serverFile` component are handled immediately, even during reconfiguration. The corresponding Pycots reconfiguration script contains about 30 lines of Python (without taking into account the code of the new implementation classes).

6 Related work

OpenCOM [15], Fractal [4] and FraSCAti [17] are well-known component models with similar capabilities for managing and reconfiguring component assemblies at runtime. For each component, controller elements are responsible for managing the reconfiguration operations and ensure their safety and consistency. To achieve these guarantees, components can be stopped such that they are led to a *quiescent state*. A typical reconfiguration scenario is (1) stop affected components, (2) change bindings, and (3) (re)start components. Unaffected components are not stopped hence their services remain available during reconfiguration.

When a component A attempts to use a stopped component B, while the behavior is said undefined in the Fractal textual description, most implementations suspend the calling thread until B is restarted. Even if A is not explicitly stopped, its services are unavailable and unavailability propagates back in the architecture. Alternatively, in the Boyer et al.'s work [2], a consistency invariant requires that mandatory dependencies of started components are bound to started components only: in this case, A must be stopped before B can be. While this approach is better founded, in practice applications are often stopped entirely. OSGi proposes yet another alternative: the framework-provided `getService` method, used by a bundle to resolve a dependency, informs the bundle when the dependency is missing. To some extent, OSGi supports only optional bindings, which is hard to satistify in practice.

The proposal of Bialek and Jul [1] envisions to facilitate the reconfiguration of component-based distributed applications. To take into account the requirement of non-stopping components while reconfiguring, they maintain at the same time the previous and the new versions of a component that needs to be changed. This strategy introduces complexity for managing these elements and may bring up scalability issues. Moreover, the proposal lacks a strong formalism that would ensure important properties throughout the reconfiguration process, such as consistency.

The position paper of La Manna [12] proposes to model the current and new versions of the components using interface automata. These models are then used to automatically generate state transformers, which are functions that map states between the two versions of the interface automata. A state transformer tells when a component can switch from its current version to its new version. This promising approach provides timely, not-disruptive and safe reconfiguration, but the proposal does not consider the implementation aspects.

In summary, although the cited approaches support the dynamic reconfiguration of component-based applications, most of them do not address the requirement of service continuity while performing the reconfiguration actions. Unlike the proposals discussed in this section, our approach relies on DSU instead of the conventional start/stop operations. Coqcots focuses on maintaining safety and consistency throughout a reconfiguration process and the Coq proof environment allows to alleviate the complexity of performing DSU operations and enforce properties. Thus, correctness properties and service continuity can be proved by using this approach.

7 Conclusion

Dynamic reconfiguration provides a solution when stopping a component-based software system is not an option. Unlike previous work, our proposal relies on DSU to avoid the conventional start/stop operations over components.

Specific component implementations are used during reconfiguration in order to continuously provide the best possible service. These implementations do not need to be anticipated at design time as DSU let us embed them in the reconfiguration. In this paper, we support verification and validation aspects with Coqcots using the Coq proof assistant. By forcing the reconfiguration developer to explicitly reflect any service degradation in the type of the components, Coqcots makes service continuity controllable and provable. We also describe Pycots, an implementation framework developed using the Python language and the Pymoult library. Our case study demonstrates the advantages of the approach.

Currently, Coqcots and Pycots are still independent as passing from one framework to the other is done by hand. In future work, we plan to better integrate them by generating the Coqcots model of the architecture from the running software system, then translating automatically the Coqcots reconfiguration script to Python. The former will need to add introspection support to Pycots and the Coq's extraction mechanism paves the way to the latter. We also plan to improve the reconfiguration language, which is currently based on Coq, and define specific Coq notations and tactics to hide low-level details of Coqcots such as frame axioms.

References

[1] R. Bialek and E. Jul. A framework for evolutionary, dynamically updatable, component-based systems. In *Proc. of the Workshops at the 24th International Conference on Distributed Computing Systems*, ICDCS 2004, pages 326–331, USA, 2004. IEEE.

[2] F. Boyer, O. Gruber, and D. Pous. Robust reconfigurations of component assemblies. In *Proc. of the 35th International Conference on Software Engineering*, ICSE'13, pages 13–22, Piscataway, NJ, USA, 2013. IEEE Press.

[3] P. Brada. Enhanced type-based component compatibility using deployment context information. *Electronic Notes in Theoretical Computer Science*, 279(2):17–31, Dec. 2011.

[4] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The FRACTAL component model and its support in Java. *Software: Practice and Experience*, 36(11-12):1257–1284, 2006.

[5] J. Buisson and F. Dagnat. ReCaml: Execution state as the cornerstone of reconfigurations. In *Proc. of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP'10, pages 27–38, New York, NY, USA, 2010. ACM.

[6] K. Gama, W. Rudametkin, and D. Donsez. Resilience in dynamic component-based applications. In *Proc. of the 26th Brazilian Symposium on Software Engineering*, SBES 2012, pages 191–195, Piscataway, NJ, USA, 2012. IEEE.

[7] M. Ghafari, P. Jamshidi, S. Shahbazi, and H. Haghighi. An architectural approach to ensure globally consistent dynamic reconfiguration of component-based systems. In *Proc. of the 15th ACM SIGSOFT Symposium on Component-Based Software Engineering*, CBSE'12, pages 177–182, New York, NY, USA, 2012. ACM.

[8] P. J. Hayes. The frame problem and related problems in Artificial Intelligence. Technical report, Stanford University, Stanford, CA, USA, 1971.

[9] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Trans. on Software Engineering*, 16(11):1293–1306, Nov. 1990.

[10] W. Li. QoS assurance for dynamic reconfiguration of component-based software systems. *IEEE Trans. on Software Engineering*, 38(3):658–676, 2012.

[11] K. Makris and R. A. Bazzi. Immediate multi-threaded dynamic software updates using stack reconstruction. In *Proc. of the 2009 Conference on USENIX Annual Technical Conference*, USENIX'09, page 31, Berkeley, CA, USA, 2009. USENIX Association.

[12] V. P. L. Manna. Local dynamic update for component-based distributed systems. In *Proc. of the 15th ACM SIGSOFT Symposium on Component-Based Software Engineering*, CBSE'12, pages 167–176, New York, NY, USA, 2012. ACM.

[13] S. Martinez, F. Dagnat, and J. Buisson. Prototyping DSU techniques using Python. In *Proc. of the 5th Workshop on Hot Topics in Software Upgrades*, HotSWUp'13, Berkeley, CA, USA, 2013. USENIX.

[14] E. Miedes and F. D. Muñoz-Escoí. A survey about dynamic software updating. Technical Report ITI-SIDI-2012/003, Instituto Universitario Mixto Tecnológico de Informática, Universitat Politècnica de València, Valencia, Spain, May 2012.

[15] P. Pissias and G. Coulson. Framework for quiescence management in support of reconfigurable multi-threaded component-based systems. *IET Software*, 2(4):348–361, 2008.

[16] H. Seifzadeh, H. Abolhassani, and M. S. Moshkenani. A survey of dynamic software updating. *Journal of Software: Evolution and Process*, 25(5):535–568, Apr. 2012.

[17] L. Seinturier, P. Merle, R. Rouvoy, D. Romero, V. Schiavoni, and J.-B. Stefani. A component-based middleware platform for reconfigurable service-oriented architectures. *Software: Practice & Experience*, 42(5):559–583, May 2012.

[18] Y. Vandewoude, P. Ebraert, Y. Berbers, and T. D'Hondt. Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates. *IEEE Trans. on Software Engineering*, 33(12):856–868, Dec. 2007.